UNIVERSIDAD POLITÉCNICA
DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE –
EUROPEAN MASTER IN SOFTWARE ENGINEERING

Development of the Infrastructure for a Multi-agent Traffic Management System Simulation

Master Thesis

Eleonora Vasileva Adova

Madrid, June 2015

This thesis is submitted to the ETSI Informáticos at Universidad Politécnica de Madrid in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering.

Master Thesis
Master Universitario en Ingenieríadel Software – European Master in Software Engineering
*Thesis Title:*Development of the Infrastructure for a Multi-agent Traffic Management System Simulation
Thesis no: EMSE-2015-02
June 2015

*Author:*Eleonora Vasileva Adova
Bachelor of Science in Computer Science
Sofia University "St. Kliment Ohridski"

Supervisor:

Ricardo Imbert
Ph.D. in Computer Science
Universidad Politécnica de Madrid

Languages, Systems and Software Engineering Department
Computer Software and Engineering School
Universidad Politécnica de Madrid

ETSI Informáticos
Universidad Politécnica de Madrid
Campus de Montegancedo, s/n
28660 Boadilla del Monte (Madrid)
Spain

Contents

Figures

I. Abstract

This document contains detailed description of the design and the implementation of a multi-agent application controlling traffic lights in a city together with a system for simulating traffic and testing. The goal of this thesis is to design and build a simplified intelligent and distributed solution to the problem with the traffic in the big cities following different good practices in order to allow future refining of the model of the real world.

The problem of the traffic in the big cities is still a problem that cannot be solved. Not only is the increasing number of cars a reason for the traffic jams, but also the way the traffic is organized. Usually, the intersections with traffic lights are replaced by roundabouts or interchanges to increase the number of cars that can cross the intersection in certain time. But still there are places where the infrastructure cannot be changed and the traffic light semaphores are the only way to control the car flows.

In real life, the traffic lights have a predefined plan for change or they receive information from a centralized system when and how they have to change. But what if the traffic lights can cooperate and decide on their own when and how to change?

Using this problem, the purpose of the thesis is to explore different agent-based software engineering approaches to design and build a non-conventional distributed system.

From the software engineering point of view, the goal of the thesis is to apply the knowledge and use the skills, acquired during the various courses of the master program in Software Engineering, while solving a practical and complex problem such as the traffic in the cities.

II. Background

Agents and multi-agents systems are often used for more complex problems where a solution of one agent or a simple system is not applicable. One definition of an agent is *"a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its delegated objectives"* [1]. A multi-agent system is a system which consists of several interacting agents and is suitable for complex distributed applications, usually used in robotics, traffic control and coordination, science simulations.

There are a lot of methodologies for the development of multi-agent systems that provide models, methods and techniques to facilitate all the phases of the development process. Some of the most common ones are MaSE, Gaia and Prometheus.

1. MaSE

Multi-agent Systems Engineering (MaSE) [2] is a methodology that covers the complete life-cycle of the development process of a multi-agent system. MaSE is an agent-oriented methodology in which the agents are simple software processes which cooperate to obtain a goal. MaSE has two phases – Analysis and Design, each of which consists of several other steps as shown in the following figure. All the steps are implemented by the graphical tool agentTool.

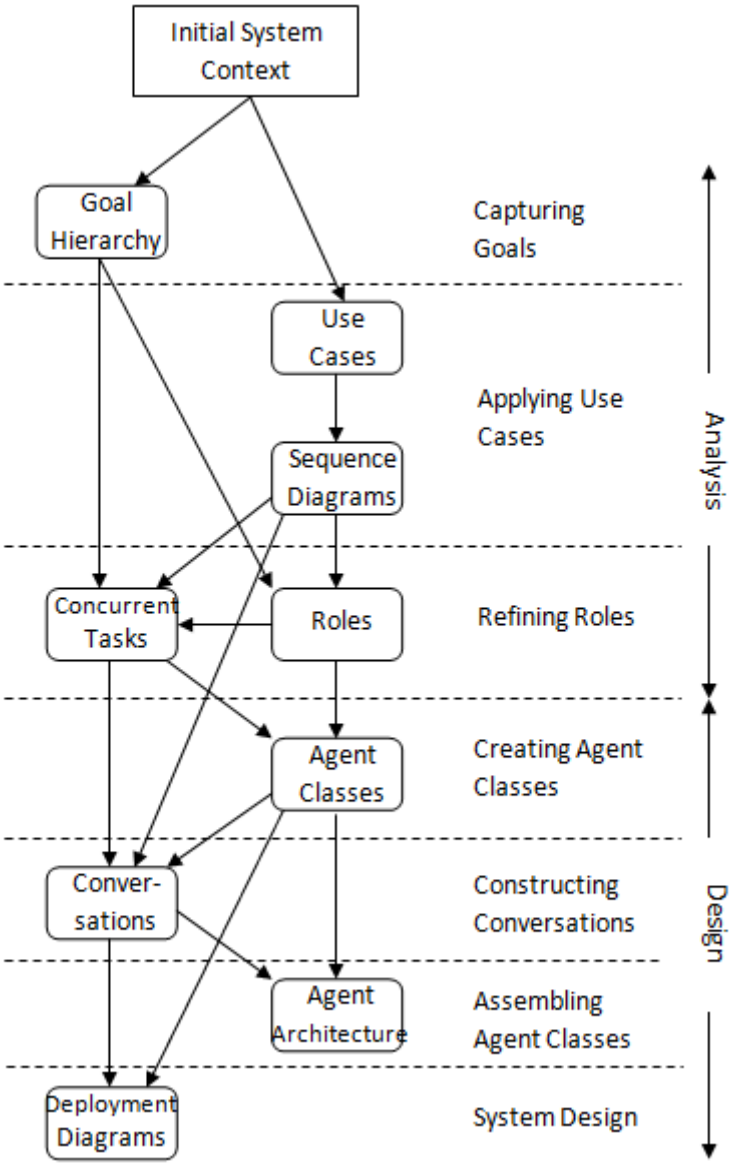

Figure 1: MaSe Phases and Steps

2. Gaia

Gaia provides a modelling framework and several associated techniques to design agent-oriented systems. In it the multi-agent system is a computational organisation which consists of various interacting roles [3].

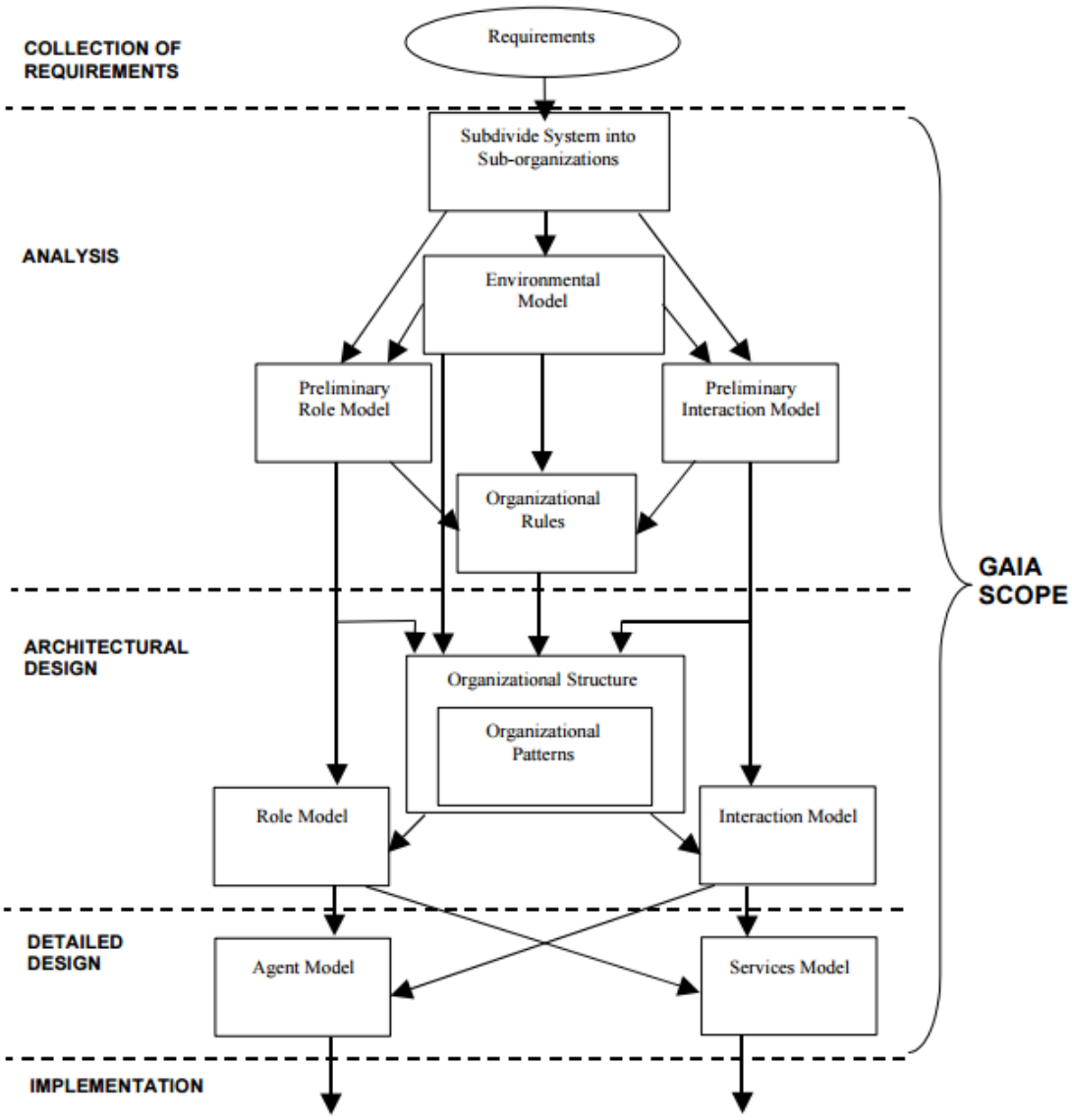

Figure 2: Models in Gaia

3. Prometheus

Prometheus is methodology which covers three phases of the software engineering process – system specification, architectural design, detailed design, and it is a mixture of graphical notation and structured text notation. It considers the development of intelligent agents in terms of goals, beliefs, plans, and events.

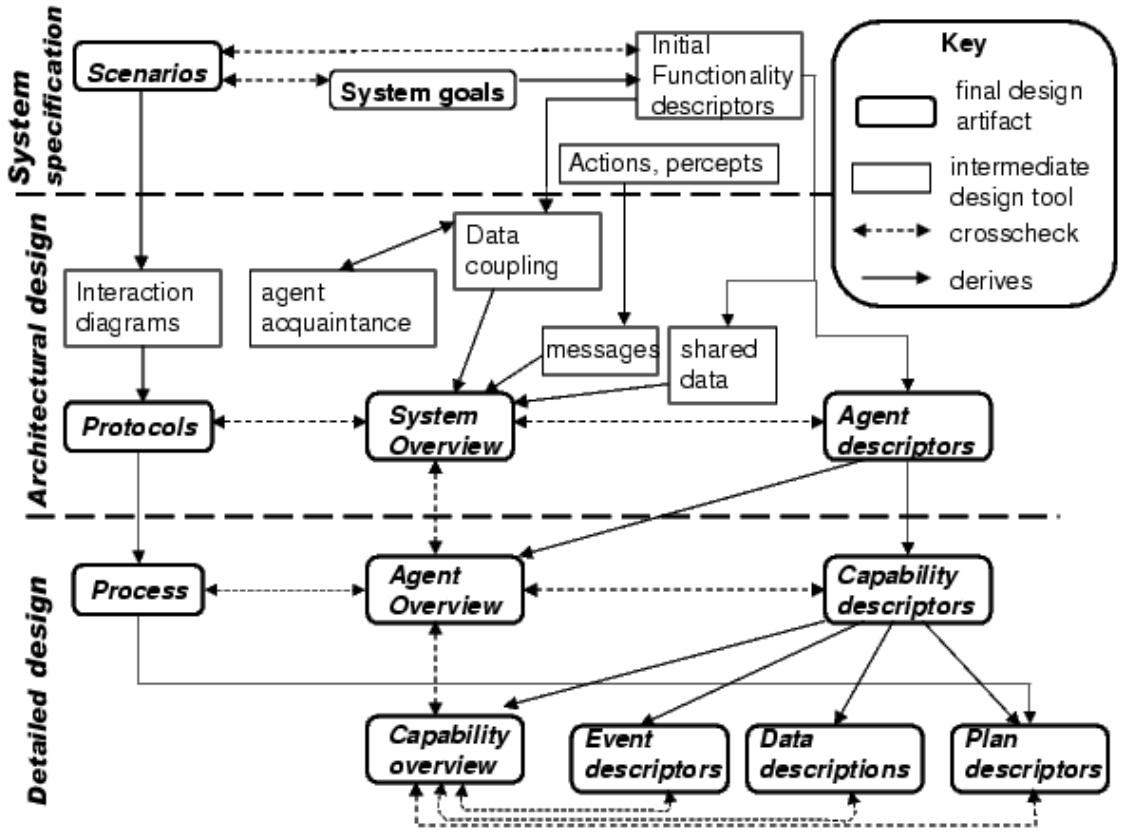

Figure 3: Phases in Prometheus

4. Methodologies Used in the Thesis

None of the methodologies has proven to be the best and all of them have advantages and disadvantages. That is why the current thesis does not follow one particular methodology, but it rather uses different tools and techniques from all of them which are found useful and suitable to the problem being solved.

As common to several methodologies, one of the steps in the analysis phase is defining the goals. Since the agents are driven by goals, it is important to define them before doing the design. In this thesis the goals are defined as suggested in Prometheus in two steps – define initial goals, and refine them.

As a next step in the analysis the roles in the system are determined following MaSE guidelines. This step is important to identify later the agents and distribute the responsibilities between them.

In the detailed design phase, the agents and their roles are described, together with the protocols for interchanging messages between the agents. AUML notation is used for the diagrams of the protocols [6]. Another step of the design phase is the design of the ontology. The ontology is explicit formal specification of a shared conceptualization and provides a shared and common understanding of the domain that can be communicated across the agents [11].

Other steps of the methodologies are found to be too complex for the purposes of the thesis or there is no practical usage in the current project, such as the liveness expressions in Gaia which purpose is to specify the behaviour of roles relative to each other in a system.

III. Analysis

The objective of the analysis phase is to develop an understanding of the system and its structure. In this section the goals and the roles of the system are defined.

1. Goals

The goals of the system have been defined as follows:

1) Minimize the average time each car spends to cross the city;
    1.1) Synchronize and manage the state of each semaphore in the city;
        1.1.1) Obtain the state of the other semaphores on the crossroad;
        1.1.2) Inform about the state of a semaphore;
    1.2) Monitor traffic level for a semaphore;
        1.2.1) Obtain number of cars in a queue for the semaphore;
        1.2.2) Obtain number of cars coming from the previous semaphores;
    1.3) Agree for the schedule for changing the lights;
2) Simulate the real world;
    2.1) Simulate traffic flow;
        2.1.1) Generate cars;
        2.1.2) Move cars to random destinations;
        2.1.3) Represent a queue for a semaphore;
    2.2) Simulate devices for counting cars;
        2.2.1) Count number of cars in a queue;
        2.2.2) Count number of cars passing;
2. Evaluate the performance of the TCMS;
    2.1) Obtain the time a car spends to cross the city.

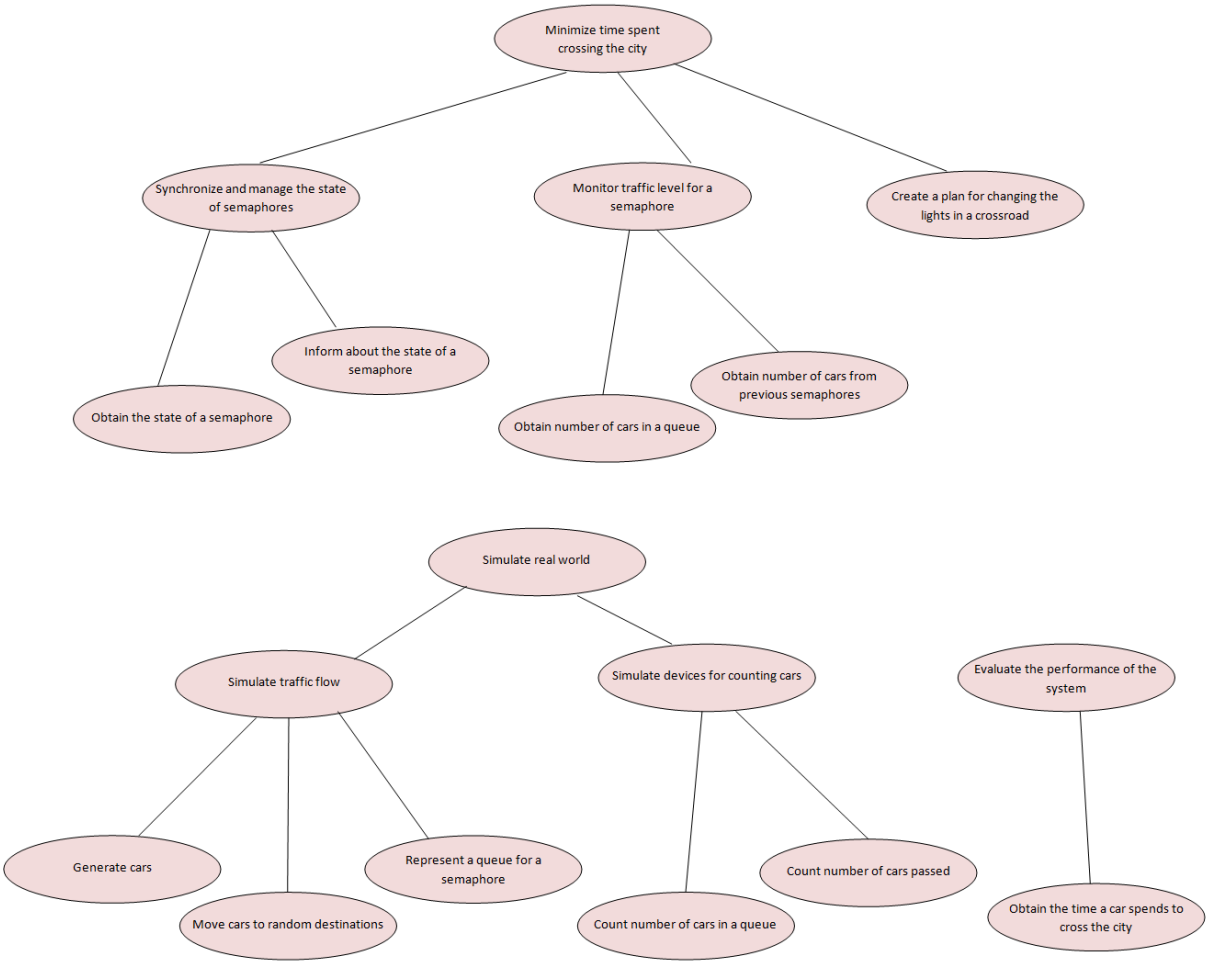

Figure 4: Goals of the System

2. Roles

After identifying the goals of the system, the following roles in the system were defined:

- Semaphore –represents one semaphore on a crossroad. It has a state which is changed by the semaphore and can be obtained.
- CarCounter – this role is responsible for counting cars waiting or moving to a crossroad.
- Car – represents a single car that moves from one direction or another in the city.
- CarGenerator – the responsibility of this role is to simulate car flows.

- Reporter – this role is responsible for gathering information for the performance of the system.

IV. Design

The project is a divided in several separate subsystems:

- Traffic Control Management System (TCMS) – the intelligent system that represents the control of the semaphores in the real world;
- Simulation System – the system for simulating the traffic and the physical devices required by the Traffic Control Management system to function;
- Graphical Application – an application with graphical user interface for visualization of the system.

In this section the design of each subsystem is described in terms of agents, interfaces, and protocols of communication.

1. Traffic Control Management System

The Traffic Control Management System (TCMS) is an independent implementation of an intelligent multi-agent system for controlling real semaphores on the crossroads of a city. The purpose of this system is to control the semaphores in the city in such a way that it minimizes the average time each car spends crossing it.

1.1. Agents

In order to achieve the goals, in the TCMS there are two types of agents – Semaphore and Controller. Each crossroad consists of four agents Semaphore representing the traffic light semaphores of each incoming lane, and one agent Controller, which is responsible for taking decisions how to change the time each semaphore has green light. The middle agent Controller is introduced to serve as an arbitrator for the four semaphores on one crossroad and avoid possible deadlocks when the semaphores have to negotiate for the schedule of the crossroad.

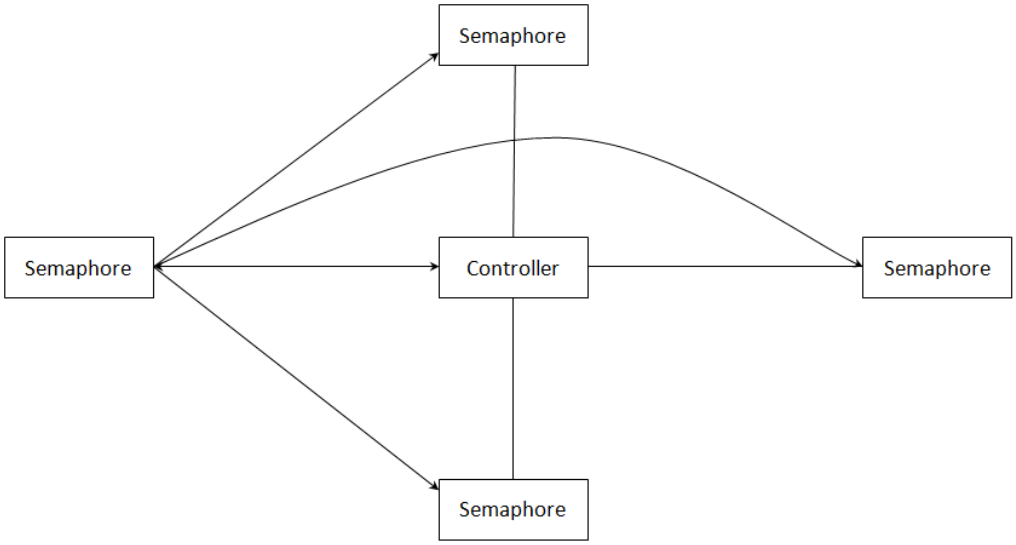

Figure 5: Agents Representing a Crossroad

Semaphore

The Semaphore is an agent that represents a traffic light semaphore for a single lane in a crossroad. It monitors the traffic of the cars passing through it and when the traffic changes it requests a change in the schedule from the controller.

Each Semaphore has a plan or schedule for how long it should stay green. Once a semaphore changes its state to *Red*, it informs all the agents that are registered for its notification of change in the state. Apart from that, each semaphore is subscribed to be notified when the states of its direct neighbours (the semaphores on the same crossroad) change. When a semaphore is in state *Red* and it receives that its left and right direct neighbours have changed their state also to *Red*, the current semaphore moves in state *Green* for the amount of time the Controller has decided last. The semaphores across one crossroad are in state *Green* the same period of time and depending on the order of the messages sent between the agents, they change their states almost at the same time. For example, in the following figure the semaphores on the east and west sides of the crossroads are in *Green* at the same time. When they change to *Red*, the semaphores in north and south will change to *Green*.

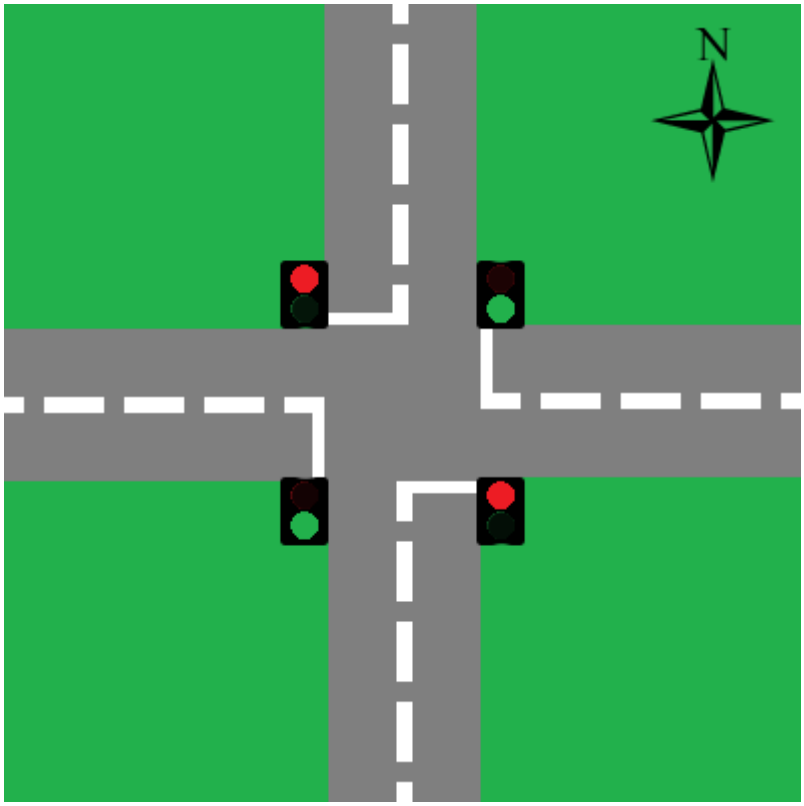

Figure 6: Semaphores in a Crossroad Open at the Same Time

The Semaphore monitors its traffic level by obtaining information about the number of cars that are already waiting for the traffic light to change to green and the number of cars that are expected to come from the three semaphores before it. In the following picture, the three semaphores in red squares affect the traffic of the one in blue.

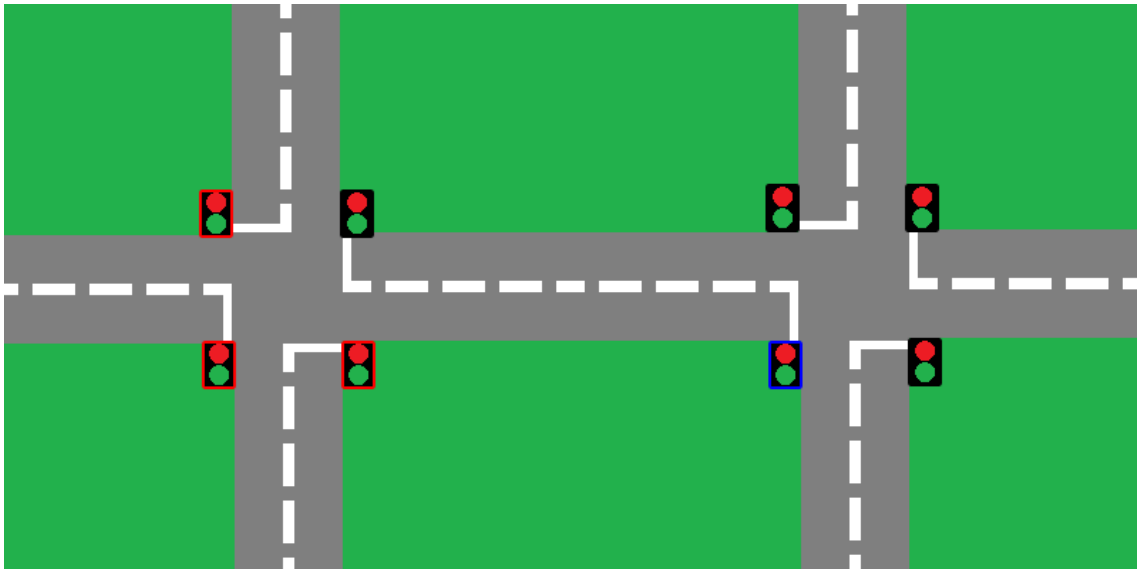

Figure 7: Semaphores Affecting One Traffic Light

The TCMS provides an interface (ICarCounter) and an implementation of the protocol *WaitingCars* as a mechanism to provide the number of the cars in a queue waiting in front of the semaphore. In the real world, this interface should be implemented by an agent which obtains this information from hardware devices, e.g. video cameras. In the simulation the interface is implemented by agents representing a queue for a semaphore. In order to make the TCMS independent and avoid coupling with the Simulation system, the name of the agent is provided when initializing a semaphore agent.

The level of the traffic is calculated by formula depending on the number of coming cars and has several states: *NoTraffic, Low, Normal, Intensive, High, and Blocked*. When the level of the traffic increases or it decreases but before was at least *Intensive*, the semaphore request from the controller to recalculate and increase its time in state Green, taking into account the traffic of the other semaphores.

Number of Cars	Traffic Level
0	*NoTraffic*
< 5	*Low*
< 10	*Normal*
< 20	*Intensive*
< 30	*High*
>= 30	*Blocked*

Figure 8: Traffic Level Corresponding to Number of Cars

Each Semaphore agent needs the names of the other semaphores on the same crossroad with their directions (*Left*, *Right*, and *Front*), and the name of its controller. The local names are retrieved in the setup of the semaphore.

Controller

The Controller is an agent responsible for scheduling the changes in the state of the semaphores on one crossroad. Each crossroad has

only one controller and each semaphore in the crossroad is registered in it.

When a controller receives a request from a registered semaphore, it asks all the semaphores on the crossroad for their level of the traffic and information about the expecting cars. Using this information the controller calculates a schedule with certain strategy algorithm which tries to minimize the time each car waits on the crossroad. The schedule consists of the time interval in which each pair of semaphores will be in state *Green*. Then the controller informs the semaphores for its decision and it is applied on the next change.

1.2. Protocols

This section contains detailed description of the protocols used by the agents to exchange messages in the TCMS. The protocols follow the FIPA standards, and apart from the description, each one is presented by an AUML diagram [6] and a table with the following information:

- Protocol name;
- Initiator;
- Partner;
- Performative;
- Immediate response;
- Later response;
- Content of the message (messages)

1.2.1. Initial Registration

At the beginning each semaphore registers in the controller of the crossroad. After all the semaphores have been registered the controller starts to function and schedule the time of the semaphores.

Protocol Name:	InitialRegistration
Initiator:	Semaphore
Partner:	Controller
Performative:	REQUEST
Immediate Response:	AGREE, REFUSE, NOT_UNDERSTOOD
Later Response:	INFORM, FAILURE

Content of the message:	REQUEST: Registration AGREE: - REFUSE: - NOT_ UNDERSTOOD: - INFORM: RegistrationSucceeded FAILURE: -

The Semaphore sends a request for the registration to the controller with its name and the name of the semaphore which will be in state *Green* at the same time. The name of this semaphore (the opposite semaphore of the initial one) is retrieved by the object parameters when initializing the Semaphore agent.

If the semaphore is already registered or the pair of semaphores does not match any of the previous registrations, the controller refuses the request. If the format of the message is not correct the semaphore responds with NOT_UNDERSTOOD. Otherwise the controller sends AGREE and registers the semaphore. After that sends INFORM to notify that the request is completed.

All the semaphores should be registered before the controller starts to manage their schedule. If a semaphore is not registered, later it cannot requests changes in the schedule.

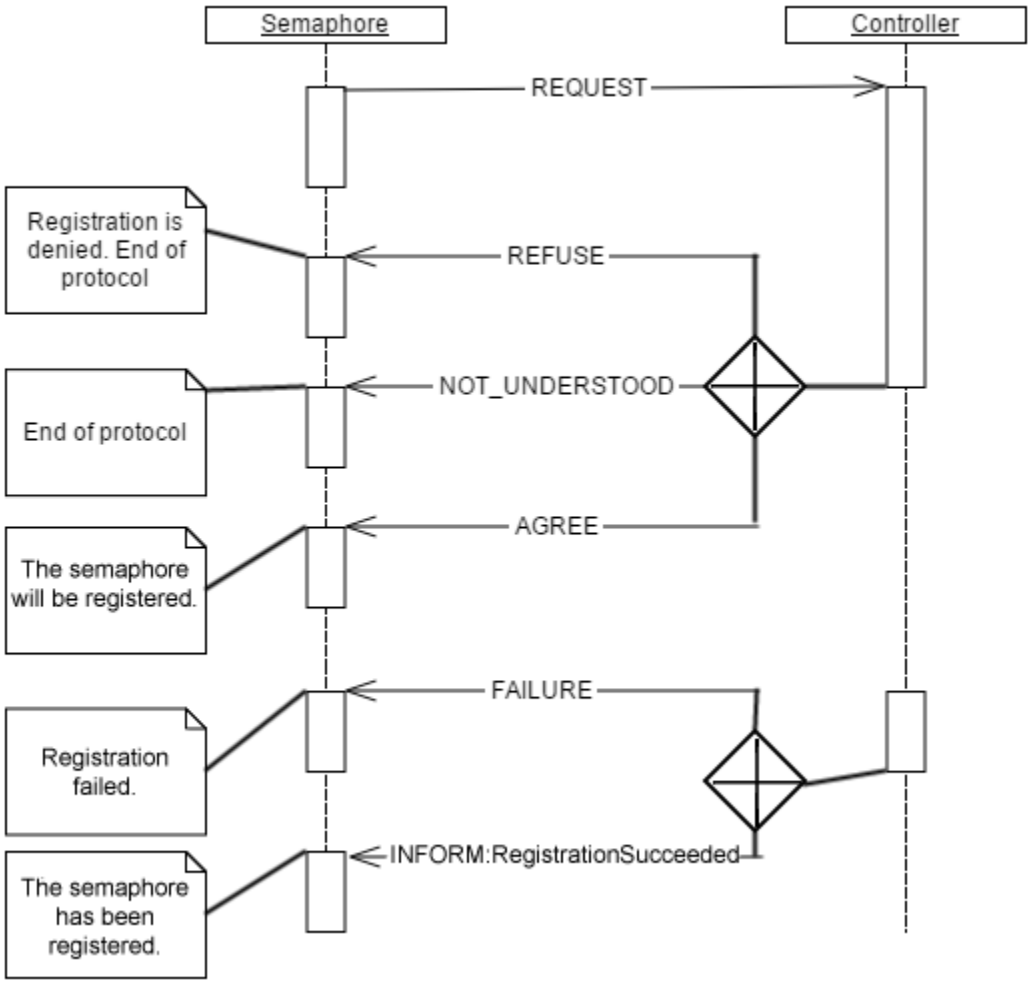

Figure 9: Initial Registration Protocol

1.2.2. State Subscription

This protocol is used by different type of agents to subscribe for notifications when the state of a semaphore changes.

Protocol Name:	StateSubscription
Initiator:	Agent
Partner:	Semaphore
Performative:	SUBSCRIBE
Immediate Response:	AGREE, REFUSE, NOT_UNDERSTOOD
Later Response:	INFORM
Content of the message:	SUBSCRIBE: StateSubscription AGREE: - REFUSE: - NOT_ UNDERSTOOD: -

	INFORM: State

If an agent wants to be informed for the change in the state of semaphore, it sends a message with performative SUBSCRIBE and protocol name StateSubscription. If the agent is already registered, the semaphore refuses the subscription. If something is wrong with the message the response is NOT_UNDERSTOOD. Otherwise the semaphore registers the agent. When the state of the semaphore changes it sends notification to all the agents that are already subscribed.

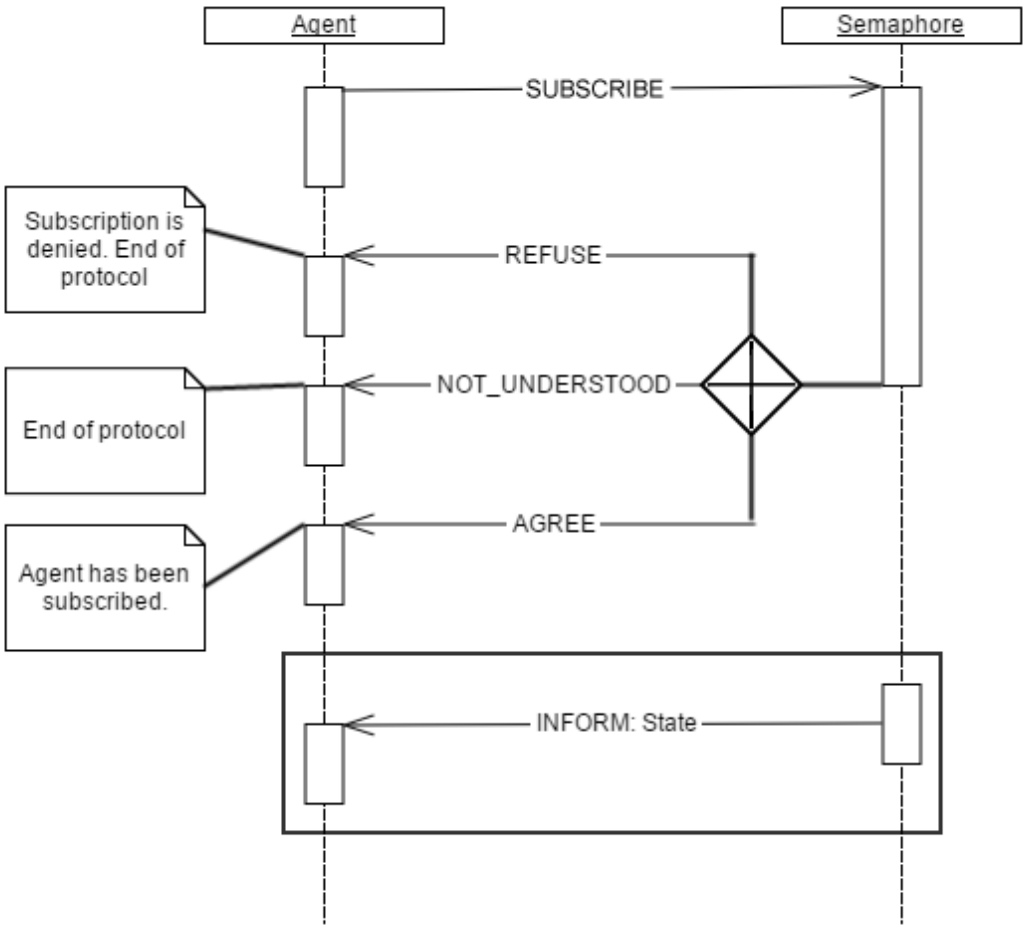

Figure 10: State Subscription Protocol

1.2.3. Request Number of Coming Cars

The Semaphore has a corresponding agent which provides information for the number of the cars waiting for the semaphore on the incoming lane. The Semaphore requests this information

periodically. Like the other semaphores of the crossroad, the name of the agent counting cars is provided in the setup parameters.

Protocol Name:	WaitingCars
Initiator:	Semaphore
Partner:	Agent
Performative:	QUERY_REF
Immediate Response:	INFORM, REFUSE, NOT_UNDERSTOOD
Later Response:	-
Content of the message:	QUERY_REF: WaitingCarsQuery INFORM: WaitingCars REFUSE: - NOT_ UNDERSTOOD: -

The Semaphore sends a QUERY_REF message to the agent responsible for counting the cars. If the message is not formatted well, the agent responds with NOT_UNDERSTOOD. Otherwise the agent sends the number of cars waiting for the semaphore. It may happen that the agent sends REFUSE if the semaphore is not allowed to request this information.

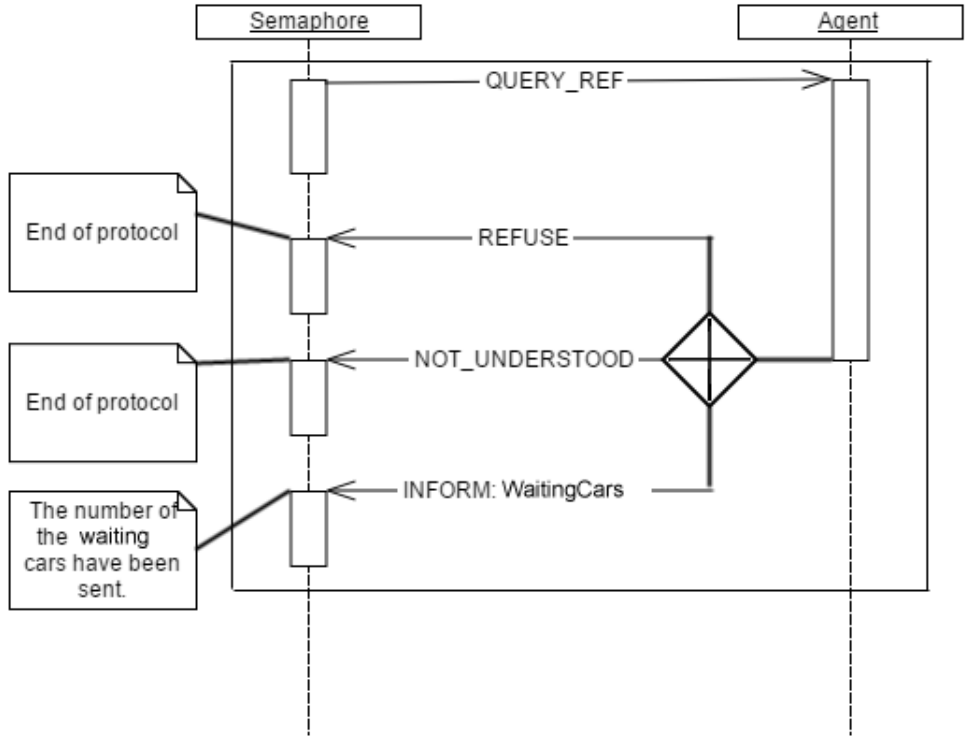

Figure 11: Waiting Cars Protocol

1.2.4. Request for Changing the Plan

Each semaphore monitors its traffic level. When it increases, or when it decreases from Intensive, High, Blocked, to any other level, the semaphore requests from the Controller of the crossroad to revise the schedule of the lights change.

This request can be done only by previously registered semaphores. Otherwise the controller refuses the request.

Protocol Name:	ChangePlan
Initiator:	Semaphore
Partner:	Controller
Performative:	REQUEST
Immediate Response:	AGREE, REFUSE, NOT_UNDERSTOOD
Later Response:	INFORM
Content of the message:	REQUEST: PlanRequest AGREE: - REFUSE: - NOT_ UNDERSTOOD: - INFORM: ChangePlan

If the controller sends AGREE, after it gets the traffic level of all semaphores, it calculates the next plan and informs all the semaphores about it.

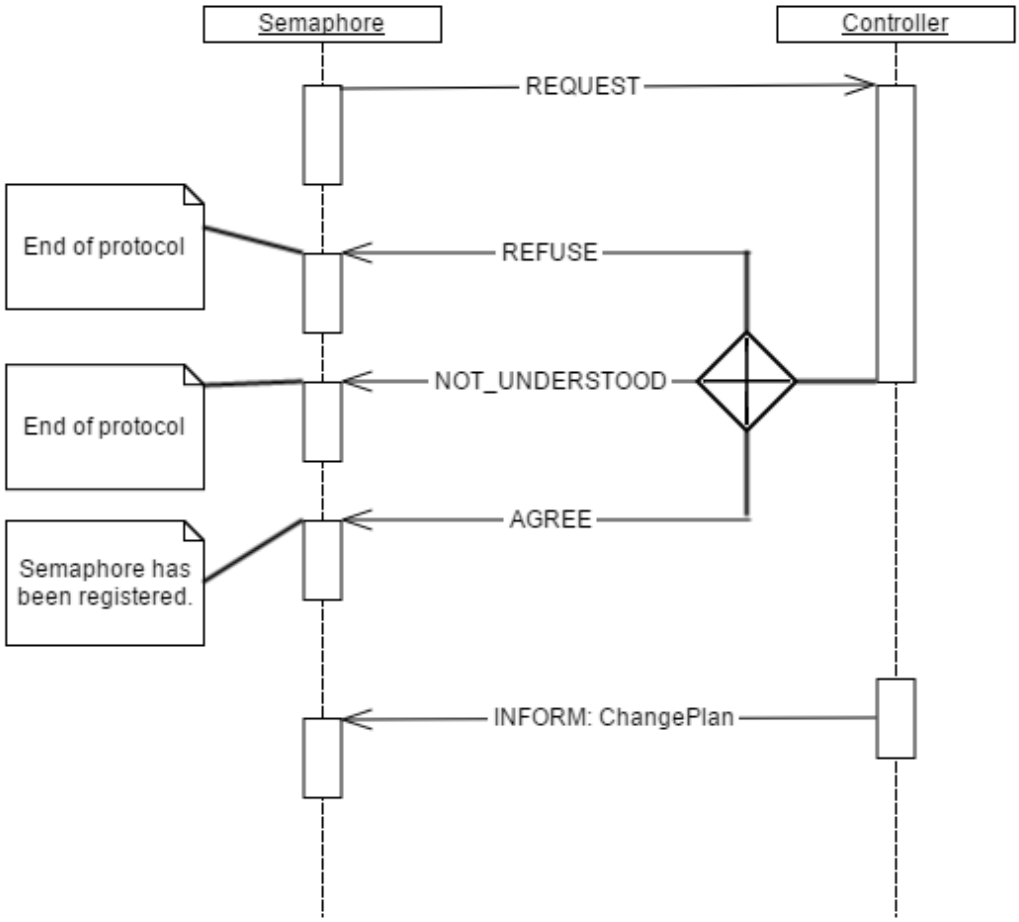

Figure 12: Change Plan Protocol

1.2.5. Request Traffic Level

When a controller receives a request to revise the time each semaphore is green, it asks all the semaphores for their traffic level. Apart from the level – Low, Normal, Intensive, etc., the semaphores send the number of cars in a queue and the number of the coming cars from their previous semaphores.

Protocol Name:	TrafficLevel
Initiator:	Controller
Partner:	Semaphore
Performative:	QUERY_REF
Immediate Response:	INFORM, REFUSE, NOT_UNDERSTOOD
Later Response:	-
Content of the message:	QUERY_REF: TrafficLevelQuery REFUSE: -

	NOT_ UNDERSTOOD: - INFORM: TrafficLevel

The controller sends a QUERY_REF message, and the semaphores answer with INFORM with the traffic information if everything is fine, REFUSE if there Controller is not the controller of the crossroad, or NOT_UNDERSTOOD if the message is not well-formatted.

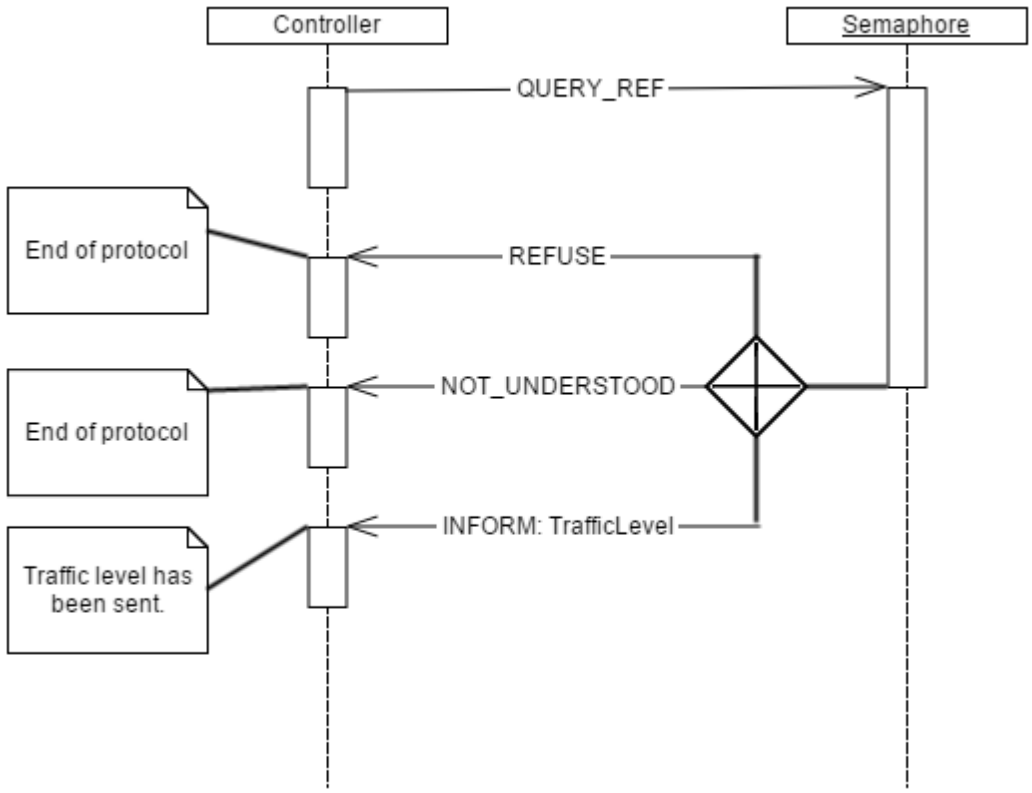

Figure 13: Request Traffic Level Protocol

1.2.6. Request Number of Coming Cars

As explained before, the level of the traffic of a semaphore is affected by the three semaphores that are positioned just before it. In order to calculate its traffic level, a semaphore asks periodically these semaphores for the number of the cars that are expected to pass the crossroad. The names of the affecting semaphores are also provided in the setup parameters.

Protocol Name:	ExpectedFlow
Initiator:	Semaphore

Partner:	Semaphore
Performative:	QUERY_REF
Immediate Response:	INFORM, REFUSE, NOT_UNDERSTOOD
Later Response:	-
Content of the message:	QUERY_REF: ExpectedCarsQuery REFUSE: - NOT_ UNDERSTOOD: - INFORM: ExpectedCars

The responding semaphore sends INFORM with number of cars which are passing the crossroad in its next turn, or sends NOT_UNDERSTOOD if the format of the message is wrong. If the semaphore has not started working it sends REFUSE.

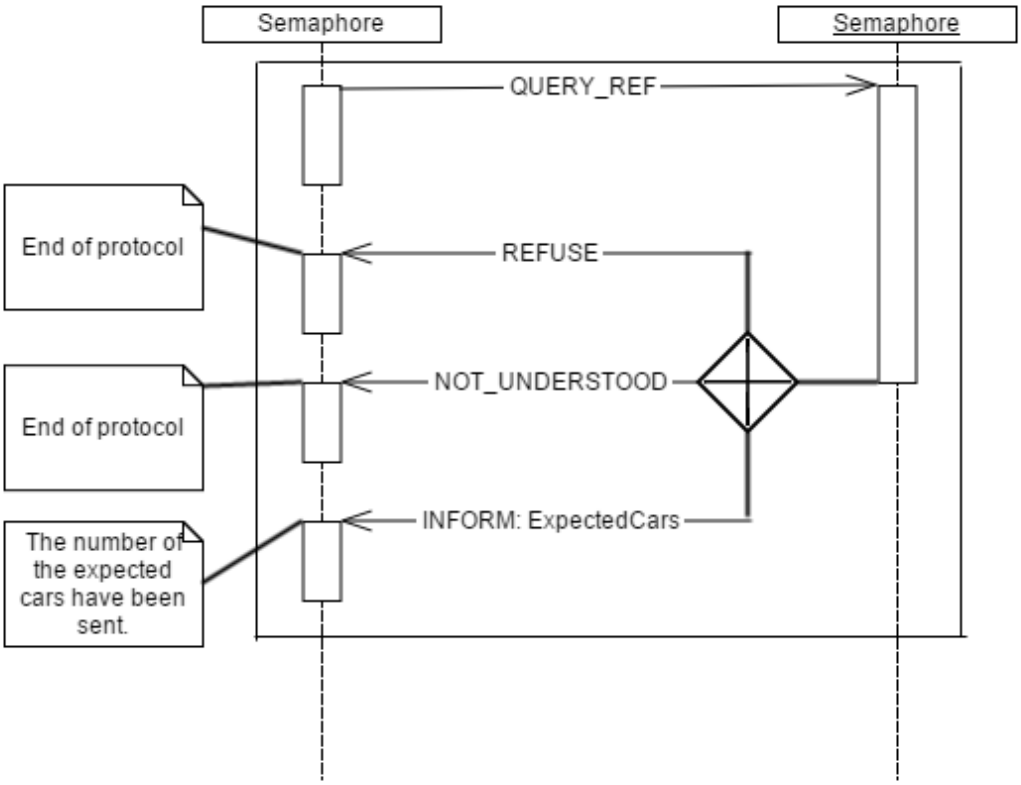

Figure 14: Request Number of Expected Cars Protocol

1.3. Ontology Design

Ontology is an explicit formal specification of how to present objects, concepts and other entities shared between agents. The protocols of the TCMS use a shared ontology which allows the following concepts, predicates and actions:

1.3.1. Concepts

Agent (agent : String)

The shared ontology has only one concept – Agent. It represents an agent in the system and contains the local name of the agent so that it can be identified with it.

1.3.2. Predicates

The predicates are expressions that say something about the status of the world and can be true or false [8].

RegistrationSucceeded ()

The predicate RegistrationSucceeded is sent when a semaphore was successfully registered in the controller of its crossroad.

State (lightState: int)

The predicate State represents the state of a semaphore – *Red* or *Green*, and corresponds to the enumeration State of the TCMS.

ChangePlan ((greenPeriod : int)

(redPeriod : int))

The predicate ChangePlan corresponds to the entity Schedule of the TCMS and contains two intervals – the time each groups semaphores in a crossroad has green lights. ChangePlan is semaphore oriented that is why the first interval is called greenPeriod, and the second one – redPeriod – the time the other semaphores have green lights.

TrafficLevel ((trafficLevel : int)

(carsInQueue : int)

(expectedCars : int))

TrafficLevel contains the information of the traffic of one semaphore – the traffic level (NoTraffic, Low, Normal, etc), the number of cars waiting for the semaphore (the cars on the street of the

semaphore), and the number of the cars that are expected to come from the other semaphores.

ExpectedCars (expectedCars : int)

ExpectedCars contains the number of cars that are expected to come from the other semaphores.

WaitingCars (waitingCars : int)

The WaitingCars contains the number of cars that are on the street of a semaphore.

1.3.3. Actions

Registration ((semaphore : Agent)

(oppositeSemaphore : Agent))

StateSubscription (agent : Agent)

PlanRequest ( )

TrafficLevelQuery ()

ExpectedCarsQuery ( )

WaitingCarsQuery()

The actions are expressions describing actions that can be performed by agents [8]. The action Registration is a request from a semaphore to a controller to be registered. In order to register, a semaphore has to specify its opposite semaphore. This is used for grouping them and later they will receive the same schedule when a new schedule is calculated.

2. Simulation System

The purpose of the Simulation system is to simulate the real world in order to test the TCMS and provide a way to evaluate its performance. It has the following tasks:

- Simulate the devices counting the number of cars in the city – provide information for the TCMS about the number of

cars in a queue for a semaphore, and the number of the cars passing through the crossroad;

- Simulate the flow of cars in the city – generate cars with random destinations which move in the city;
- Report for the performance of the TCMS – measures the average time a car spends to cross the city.

2.1. Agents

In order to achieve the goals, in the Simulation system there are several types of agents – Car, Crossroad, StreetQueue, Simulator, Reporter, and SystemLauncher. Each real crossroad consists of one Crossroad agent and eight agents StreetQueue – four with incoming cars and four with outgoing cars.

StreetQueue

This agent represents the street between two crossroads. The StreetQueue is subscribed for the change in the state of its semaphore. The Car agents are stored in a queue and when the state of the semaphore is *Green* one by one are moved through the Crossroad agent.

The StreetQueue also implements the interface ICarCounter and counts the number of the cars that are waiting for the semaphore and the number of the cars passing through the crossroad. It uses the protocol WaitingCars from the TCMS to provide the number of cars to its Semaphore.

If the StreetQueue is an entry point of the city, then the Car agent will request to be queued. Otherwise, if the car is coming from a crossroad, the Crossroad agent is the one which requests the car to be queued.

Once the car is in the queue, the StreetQueue asks the car for its next direction – *Left, Right, Ahead*. The car is moved internally inside the StreetQueue, and when it reaches the head of the queue and the semaphore is in state *Green*, the StreetQueue agent requests the Crossroad agent to move the car in its direction.

If the StreetQueue is an exit point of the city, it requests the Car to leave the city.

Crossroad

The Crossroad agent represents a real crossroad and connects the incoming and the outgoing queues of cars. It is requested by a StreetQueue agent to move a car from one to another direction. The car is moved when it is queued in the next StreetQueue agent. The crossroad can contain not more than one car coming from the same direction, and not more than two cars coming from different directions. It manages the way the cars pass through the crossroad. If car wants to make a left turn but the crossroad has a car coming from the opposite the direction, the first car should wait. And in the other case, if a car makes a left turn and has taken already the side of the crossroad for the opposite flow of cars, the cars coming from the opposite direction should wait.

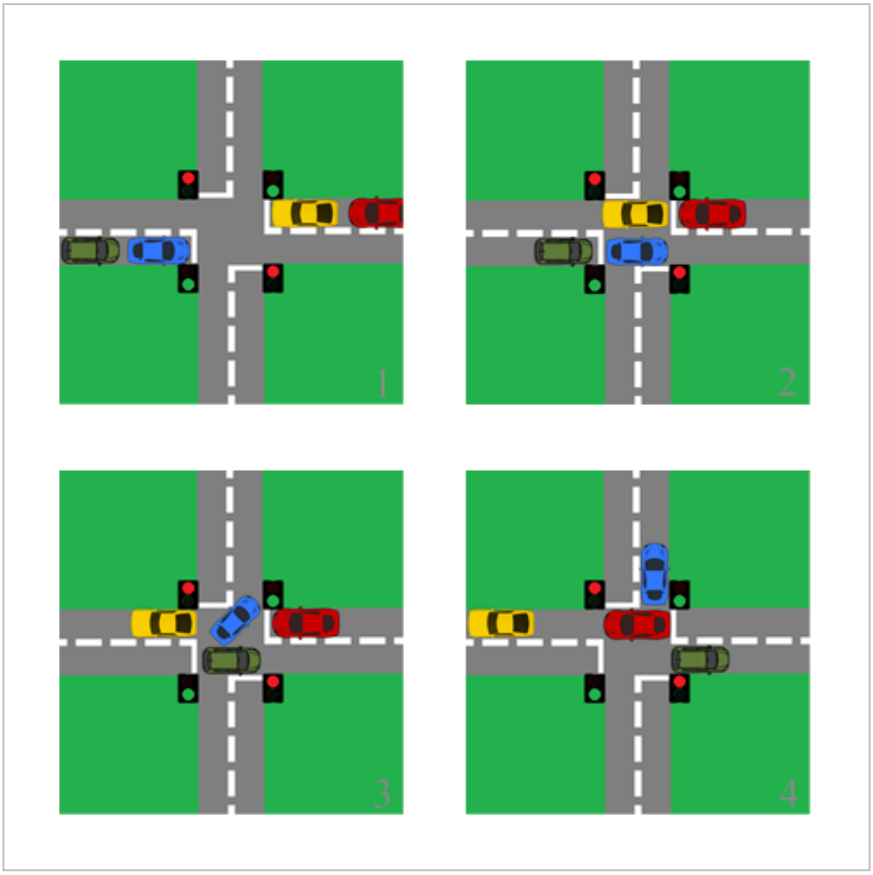

Figure 15: Crossroad Situation When a Car Waits

Unlike in the real world where the car making the left turn has to wait, here the car which has come first has the priority – even if it is the one turning to left.

Simulator

The simulator is an agent that simulates the traffic flow of the cars. It generates cars with random destinations with different intensity to simulate the different levels of traffic. The cars that are generated have different types in order to allow extension of the simulation system.

Car

The Car agent represents a single car in the city. After it is launched it enters a queue. Then it is asked for its direction several times until it reaches an exit of the city. After it is requested to leave the city, the Car agent sends a summary of the time spent to cross the city and the road it passed to the Reporter agent.

Reporter

The reporter is an agent which summarizes the execution of the simulation. The summary report provides information for the number of cars which crossed the city, and the average time divided by the distance each car spent to reach its destination.

SystemLauncher

The SystemLauncher is an agent which starts the system and the simulation. It uses a configuration file to set the settings of both subsystems and launches all their agents according to a predefined map.

2.2. Protocols

This section contains detailed description of the protocols used by the agents to exchange messages in the Simulation system.

2.2.1. EnterCity

When a Car agent is launched it requests the StreetQueue corresponding to the first street on its road to be queued and enter the city.

Protocol Name:	EnterCity
Initiator:	Car
Partner:	StreetQueue
Performative:	REQUEST
Immediate Response:	AGREE, REFUSE, NOT_UNDERSTOOD
Later Response:	INFORM, FAILURE
Content of the message:	REQUEST: QueueRequest AGREE: - REFUSE: - NOT_ UNDERSTOOD: - INFORM: CarMoved FAILURE: -

The Car sends a REQUEST message to the StreetQueue. If the queue is already full, the request is refused. If the message is invalid, the StreetQueue sends NOT_UNDERSTOOD. Otherwise the StreetQueue sends AGREE and then it tries to queue the car. If after it agrees and before it queues the car the queue becomes full, the request fails. When the car receives INFORM, it starts to count the time since it entered the city and is being transferred from one agent to another to the end of the city.

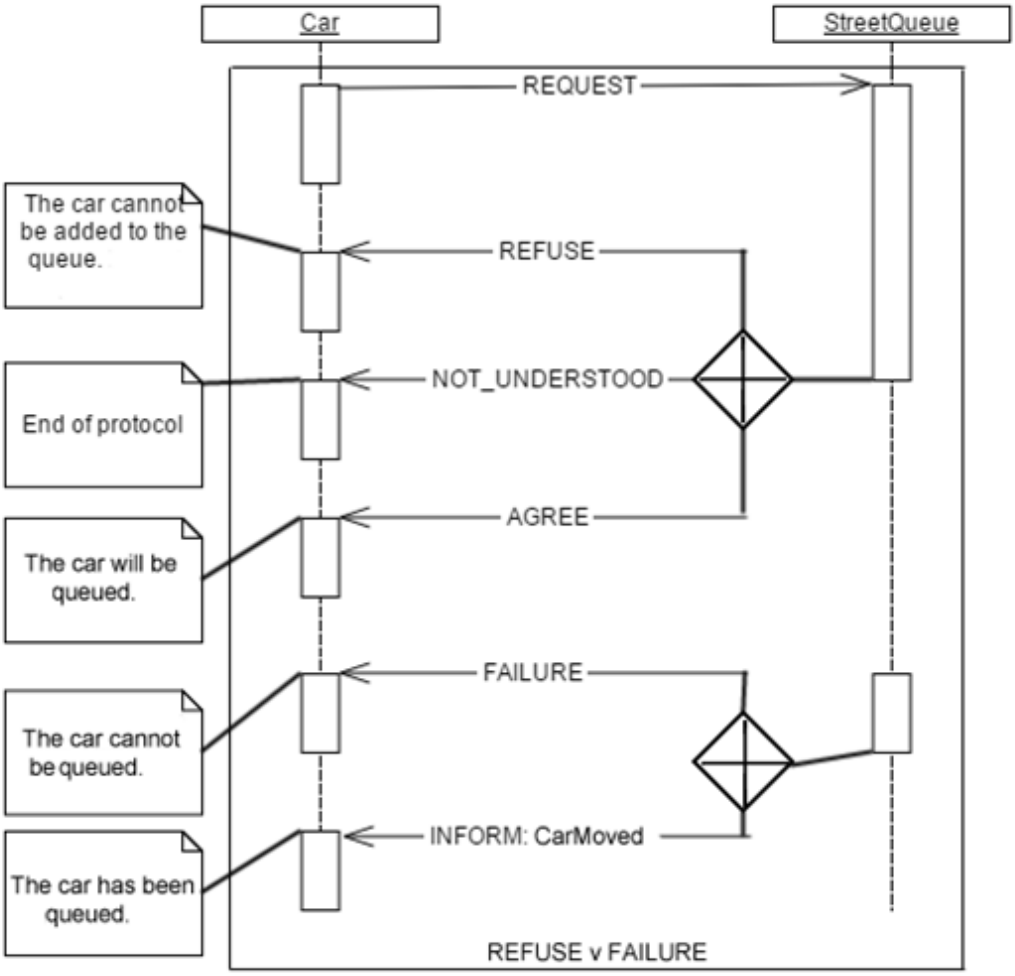

Figure 16: Enter City Protocol

2.2.2. Request Next Direction

When a car is queued in a new queue, the StreetQueue agent asks the Car for its next direction. The direction is used to move the car on the next crossroad.

Protocol Name:	RequestNextDirection
Initiator:	StreetQueue
Partner:	Car
Performative:	QUERY_REF
Immediate Response:	INFORM, REFUSE, NOT_UNDERSTOOD
Later Response:	-
Content of the message:	QUERY_REF: NextDirectionQuery REFUSE: - NOT_ UNDERSTOOD: -

	INFORM: NextDirection

If due to an error in the algorithm for generating cars or other reason the Car cannot provide next direction, it sends REFUSE. Otherwise it sends INFORM with the next direction.

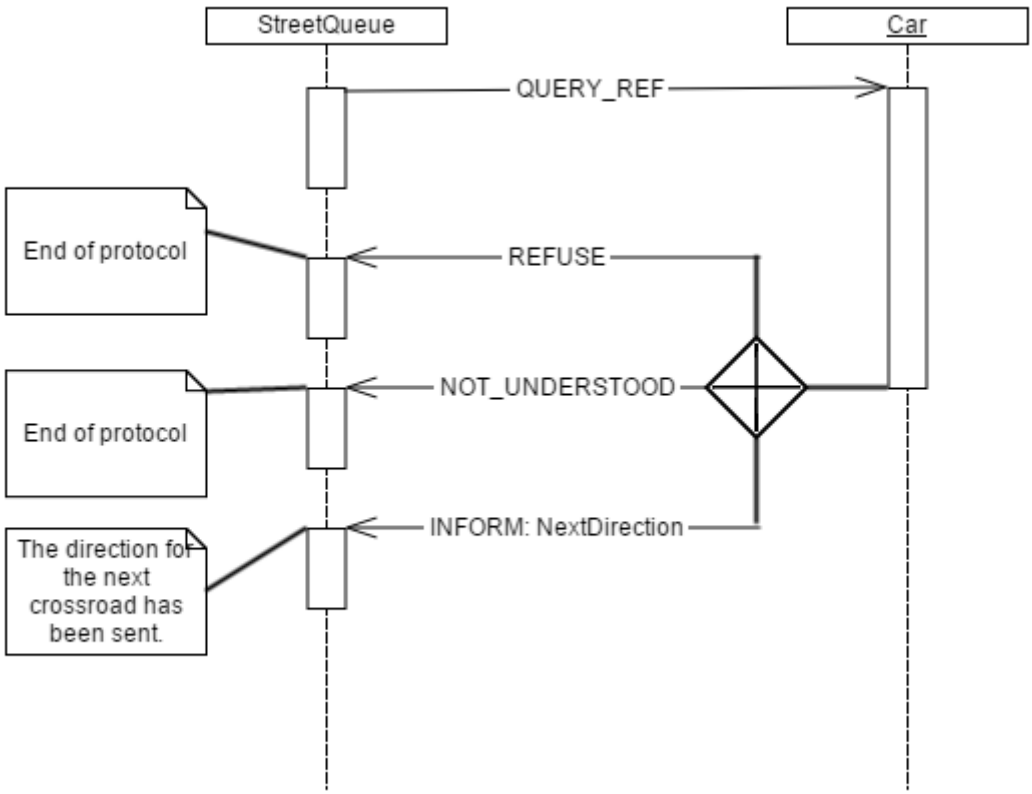

Figure 17: Request Next Direction Protocol

2.2.3. Enter Crossroad

When a car reaches the head of a queue which is a crossroad, the StreetQueue requests the Crossroad to move the Car to its next direction while the corresponding semaphore is in state *Green*.

Protocol Name:	EnterCrossroad
Initiator:	StreetQueue
Partner:	Crossroad
Performative:	REQUEST
Immediate Response:	AGREE, REFUSE, NOT_UNDERSTOOD
Later Response:	INFORM
Content of the message:	REQUEST: TransferCarRequest

	AGREE: - REFUSE: - NOT_ UNDERSTOOD: - INFORM: CarMoved

When the Crossroad receives the request if the crossroad has the maximum number of cars, it refuses the request. If the car can enter the crossroad, the response of the request is AGREE. Then after the car leaves the crossroad, the StreetQueue is informed that the request is completed.

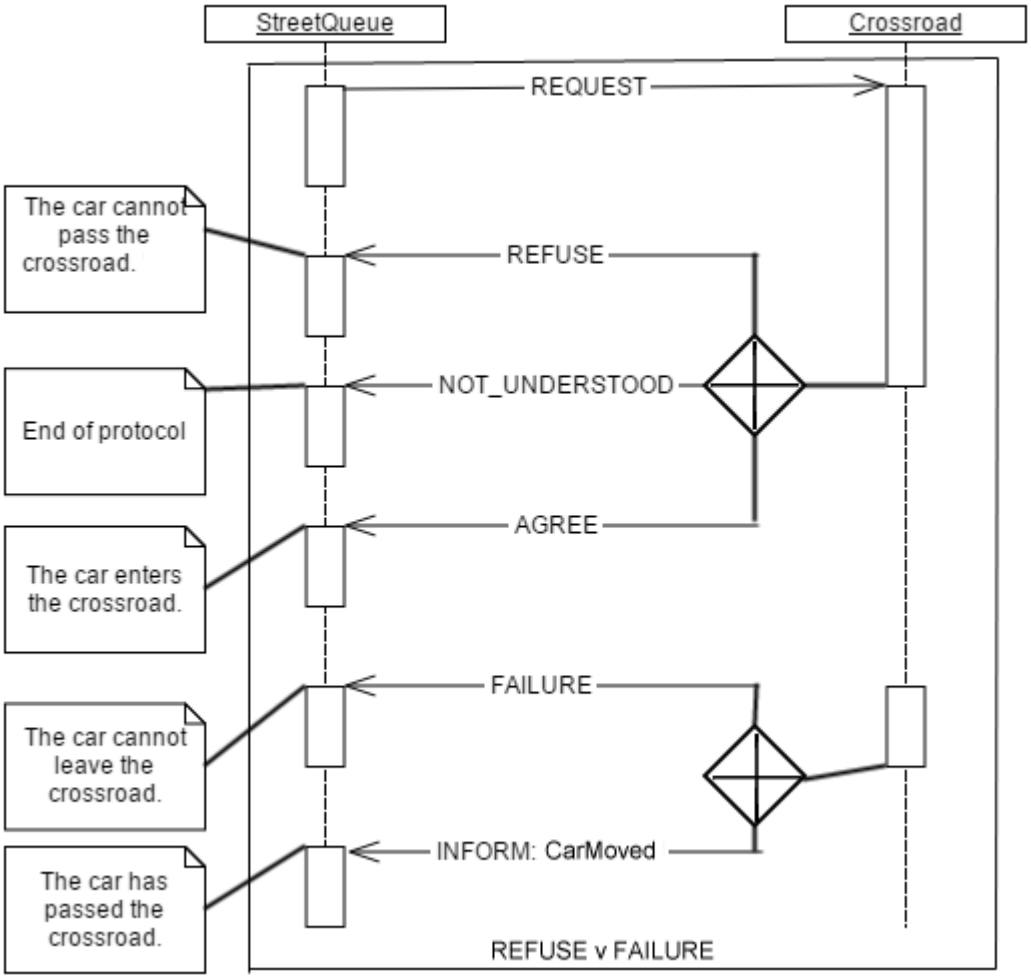

Figure 18: Enter Crossroad Protocol

2.2.4. Queue Car

When a car is on a crossroad, it may wait for a while if it is making a left turn. When the car is ready to continue on its road, the Crossroad asks the StreetQueue in the car's direction to queue the car.

Protocol Name:	QueueCar
Initiator:	Crossroad
Partner:	StreetQueue
Performative:	REQUEST
Immediate Response:	AGREE, REFUSE, NOT_UNDERSTOOD
Later Response:	INFORM
Content of the message:	REQUEST: QueueRequest AGREE: - REFUSE: - NOT_ UNDERSTOOD: - INFORM: CarMoved

The StreetQueue may refuse the request if the queue is already full. Otherwise it sends AGREE and when the request is complete, it informs the crossroad.

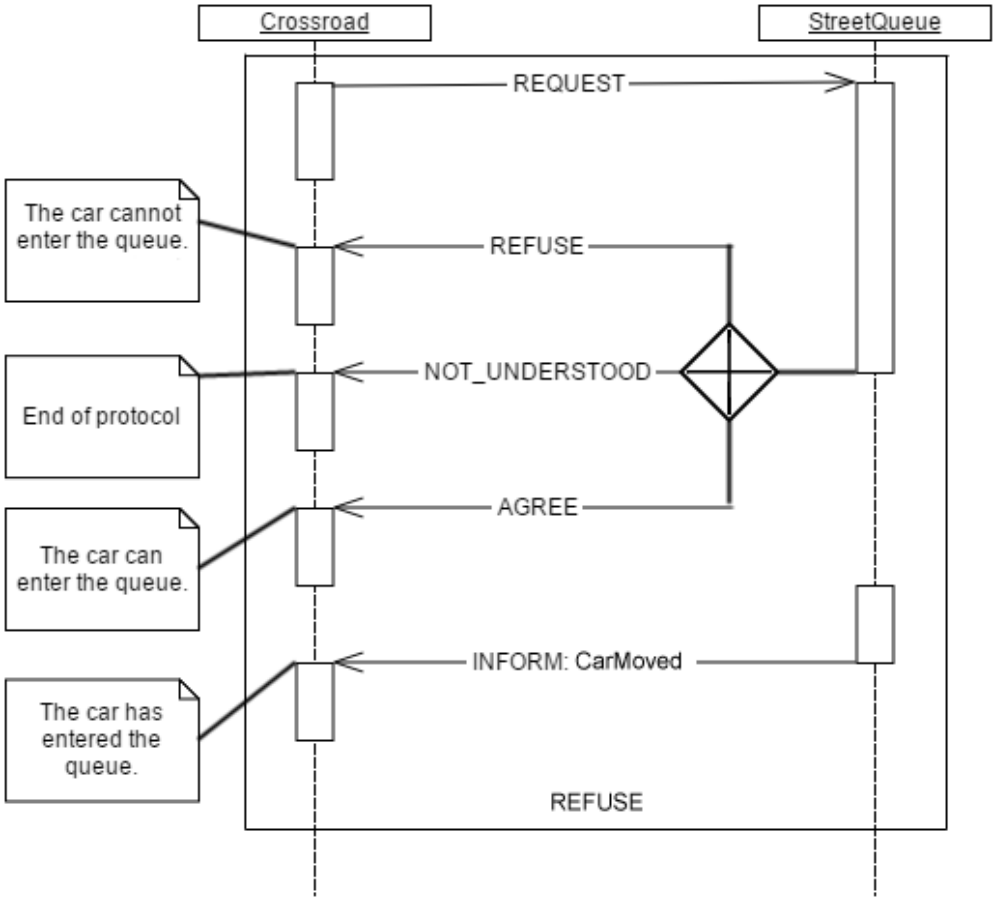

Figure 19: Queue Car Protocol

2.2.5. Leave City

When the StreetQueue does not end with a crossroad, it is an exit point of the city. If a car reaches this point the StreetQueue requests it to leave the city.

Protocol Name:	LeaveCity
Initiator:	StreetQueue
Partner:	Car
Performative:	REQUEST
Immediate Response:	AGREE, NOT_UNDERSTOOD
Later Response:	INFORM
Content of the message:	REQUEST: LeaveRequest AGREE: - NOT_ UNDERSTOOD: - INFORM: CarMoved

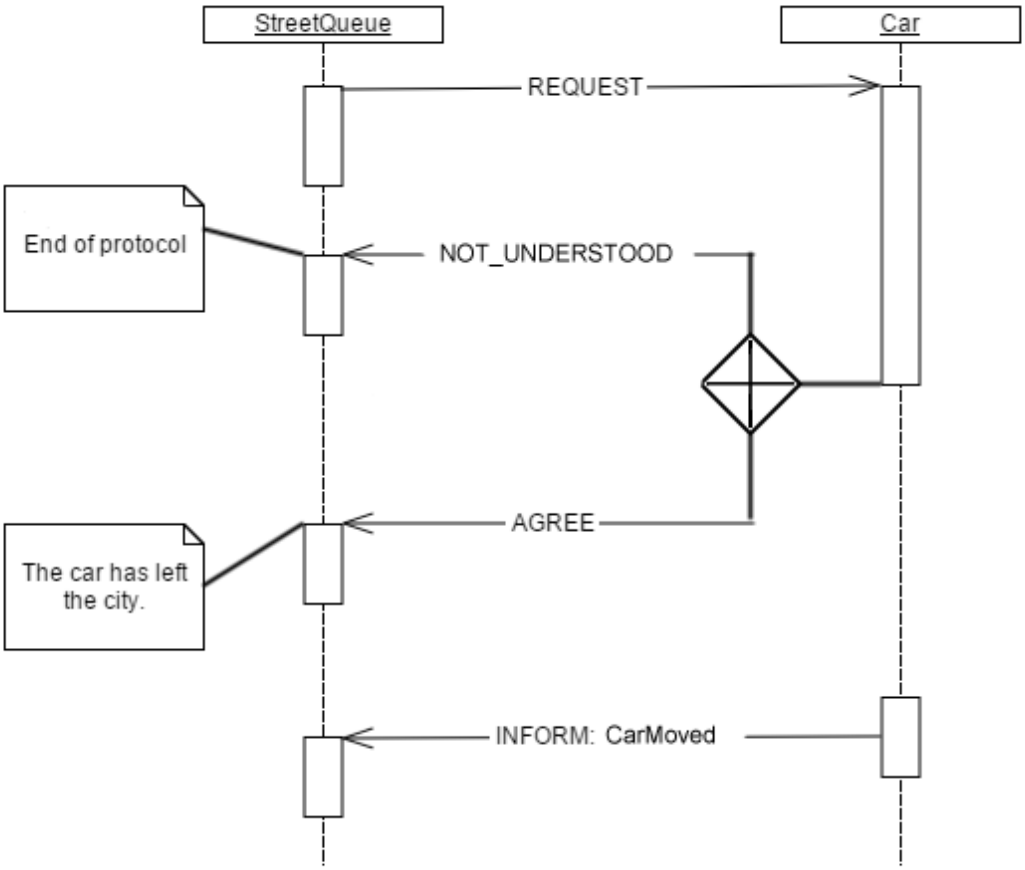

Figure 20: Leave City Protocol

2.2.6. Report Summary

After receiving a request to leave the city, the Car agent sends a summary to the Reporter for the time it spent to cross the city. The Car agent dies.

Protocol Name:	ReportSummary
Initiator:	Car
Partner:	Reporter
Performative:	INFORM
Content of the message:	INFORM: CarSummary

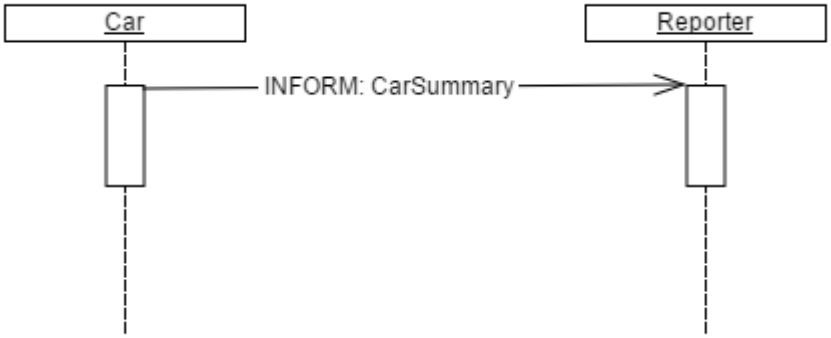

Figure 21: Report Summary Protocol

2.3. Ontology Design

The protocols of the Simulation system use a shared ontology which extends the ontology of the TCMS and allows the following concepts, predicates and actions:

2.3.1. Concepts

CarSummary ((agent : Agent)

(time : int)

(distance : int))

The concept CarSummary represents an entity with the information about one car – the time it spent to cross the city, and the distance it passed to cross the city.

2.3.2. Predicates

NextDirection (direction : int)

The predicate NextDirection contains direction and corresponds to the enumeration Direction in the TCMS.

CarMoved()

CarMoved is predicate used to inform that an operation with car has been completed and the car has moved.

2.3.3. Actions

QueueRequest (car : Agent)

NextDirectionQuery ()

TransferCarRequest ((car : Agent)

(direction : int))

LeaveRequest ( )

3. Graphical Application

As part of the thesis a simple visualization of the agents is included in order to be able to see actually how the semaphores change their lights and how the streets with more traffic have green lights longer. The visualization uses Swing and is a simple JFrame which contains visual representation of some of the agents in the system. This representation consists of drawing something – a picture, a rectangular, etc. on a place with specific coordinates.

The graphical application shows a window with the crossroads, the streets, the four semaphores on each crossroad, and the moving cars. It visualizes only agents from the simulation system, since the information about the state of the semaphores is present in the agents StreetQueue and Crossroad.

A simple view is created for each street and each semaphore. The view of a semaphore consists of showing two pictures depending on the state – a semaphore with red or green light, while the view of a street is a list of pictures with all the cars in the queue.

In order to be able to create views for the agents, all the agents of the simulation system register their instances in SystemLauncher. The registration is implemented by a callback method passed to the setup of the agents. Once all the agents are launched SystemLauncher creates a frame and draws the views for the streets and the semaphores.

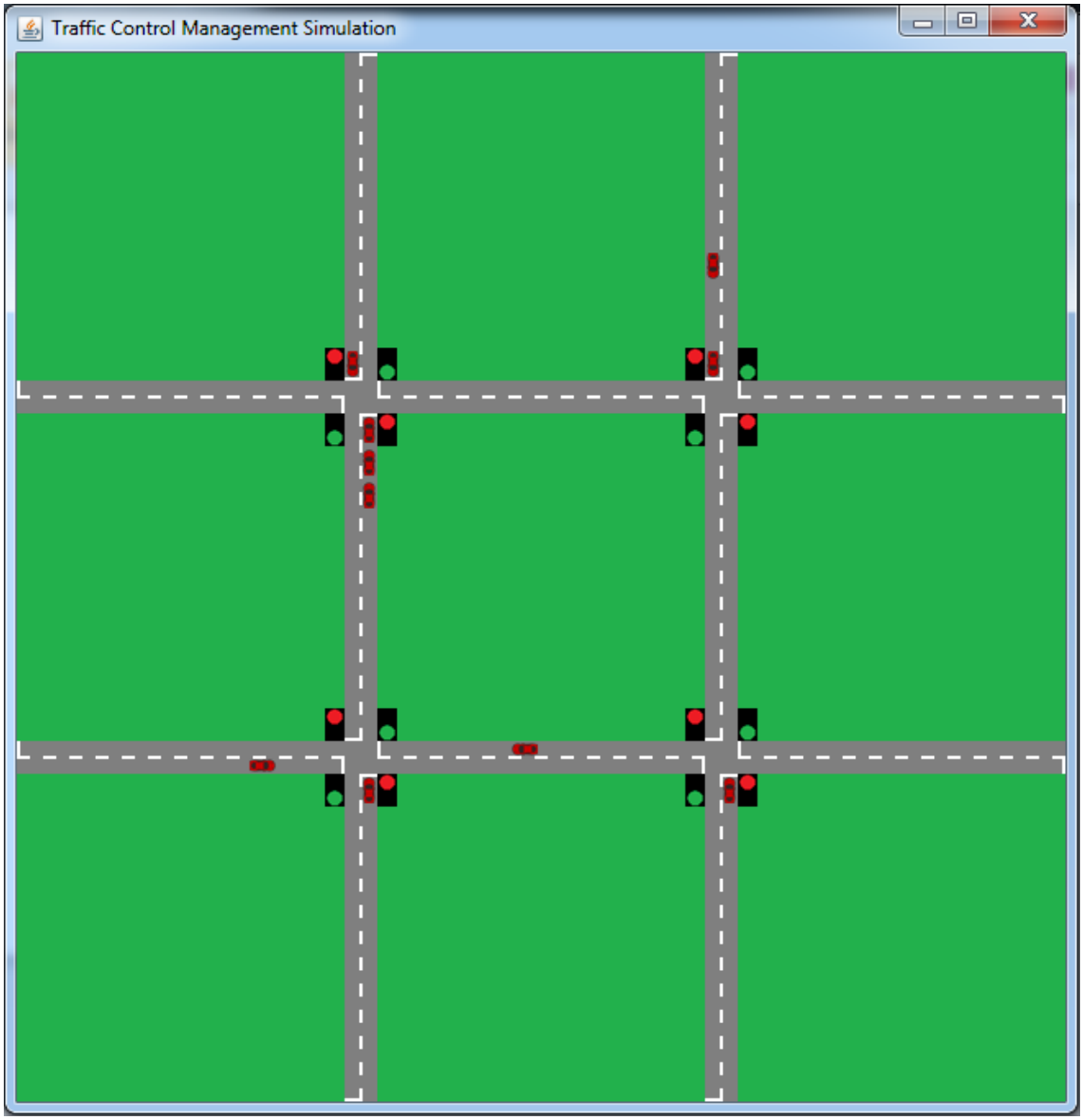

Figure 22: Graphical Application

V. Specific Implementation Details

This chapter describes specific details of the implementation of the subsystems of the project or solutions to problems which have occurred during the implementation phase.

1. Synchronization of the Semaphores on One Crossroad

One of the critical parts of the TCMS is to assure that there is no moment in which semaphores on the same crossroad, and not for opposite directions, are green. If it happens, it may lead to car accidents.

In the TCMS each semaphore is an agent and has the names of its neighbours – the other three semaphores on the crossroad. In the beginning, when the system starts, the semaphore subscribes for notifications about the state of all its neighbour semaphores.

All the semaphores start with the same schedule (30 seconds) and half of them, in couples of opposite semaphores, start with green light, and the other half start with red light. After the current schedule time for green light of a semaphore expires it changes it state to *Red*. When a semaphore is in state Red for more than certain time (5 seconds), and it receives that its left and right neighbour semaphores have changed their states to *Red*, the semaphore can and changes the light to Green. It states in this state till the time of the current schedule expires.

If a semaphore has changed its state to *Red*, it cannot go back to *Green* in the next 5 seconds. This is done in order to avoid problems caused by the delay of the messages sent between the agents and have semaphores with green lights at the same time.

2. Algorithm for Calculating the Schedule

The agent Controller, which makes the decision for the schedules that the semaphores follow, uses an algorithm which calculates the schedule according to the information for the traffic level in each semaphore on the crossroad. The strategy for making the decision is injected through the object parameters when initializing the agent with the interface *IScheduleCalculator*. The interface contains only one

method called *calculateSchedule,* which returns the schedule to be followed by the semaphores.

The schedule itself consists of two time intervals for each group of semaphores. To be able to differentiate between the different flows of cars when the semaphores register in the controller, they are split in two groups – first and second. Then when the new schedule is calculated the controller sends the first time interval to the first group of semaphores and the second time interval to the second one.

In order to be able to compare and evaluate the performance of the TCMS three different implementations of the interface *IScheduleCalculator* were created. Their description is explained in the chapter Evaluation.

3. Simulation of Hardware Devices

In the real world some hardware devices – sensors or cameras for computer vision will monitor and provide the traffic level for each semaphore. Each semaphore receives information about the number of cars coming through the protocol *WaitingCars.*

For the simulation the agent StreetQueue provides this information and has the behaviour for communicating with the Semaphore. In the real world, another agent has to be implemented. It should receive the information from the hardware devices and send it to the semaphore. When being initialized the semaphore receives only the name of the agent which implements the protocol *WaitingCars*.

4. City Map

A special class called CityMap represents the map of the streets and crossroads of the city. The class receives in the constructor a two-dimensional array of string containing the places of the crossroads.

B	S	B	S	B
S	C	S	C	S
B	S	B	S	B

B – Building (empty place)
S – Street
C - Crossroad

Figure 23: Example of the input array for CityMap

Using the input map, the map of the city is generated dynamically by splitting the streets in two lanes and the crossroads in four semaphores. The new map contains the name of the corresponding agents. The initialization also detects the entry and exit point of the city, which are used later for generating cars.

B	Q1	Q2	B	Q3	Q4	B
Q9	S1	S3	Q11	S5	S7	Q13
Q10	S2	S4	Q12	S6	S8	Q14
B	Q5	Q6	B	Q7	Q8	B

Figure 24: Example of the Internal Representation of CityMap

In the example for the internal representation of the city "S{Number}" is a name of a semaphore. The semaphores S1, S2, S3, and S4 form the first crossroad of the city. The SystemLauncher creates for them a Controller agent (C1) and a Crossroad agent (X1). "Q{Number}" is a name of a StreetQueue agent representing one lane. In the example, Q1 is an entry point of the city and its semaphore is S1, Q2 is an exit point and it does not have a corresponding semaphore.

CityMap is used by the two agents that create the other agents of the system – SystemLauncher (creates semaphores, controllers, streets, crossroads, simulator, reporter) and Simulator (generate cars).

5. Car Generation

The Simulator agent is responsible for generating car flows. It generates Car agents every five seconds for the entire execution of one hour. The simulation system has internal timer for the execution time, which starts at 09:00 a.m. and ends at 10:00 a.m. The simulator generates different number of cars at different moments.

The simulator generates different number of cars according to the current and the number of entry points of the city. The formula for the number of time is ⌈K * NumberOfEntryPoints⌉ where K is a coefficient which depends on the current time:

Time Interval	Coefficient for Number of Cars per 5 seconds
09:00:00 – 09:04:59	0.2

09:05:00 – 09:09:59	0.5
09:10:00 – 09:19:59	0.8
09:20:00 – 09:29:59	1.2
09:30:00 – 09:39:59	0.8
09:40:00 – 09:49:59	0.5
09:50:00 –	0.2

Figure 25: Coefficient for Calculating Number of Generated Cars

6. Car Movement

One of the important parts of the simulation systems is the simulation of the movement of the cars. In the project it is implemented by a class called CarQueue.

CarQueue is a special kind of queue which has capacity of 10 cars. It queues always at the same position – the last position of the queue, called tail, and it always dequeues from the same position – the first position of the queue, called head. This means that there may be a situation in which the queue cannot dequeue a car even though there are cars in the queue.

Every 570 milliseconds the queue starts from the head and tries to move each car with one position ahead. If the car cannot be moved it stays in its place. Since each street is 80 meters long and the queue has capacity of 10 cars, each 570 milliseconds a car passes 8 meters or it stays in its place. This means that the car moves with 50 km/h or does not move. Maybe if this speed is decreased, it may compensate the fact that the cars do not lose time to accelerate as they do in the real world. In this way model may become more realistic.

The following picture shows how the cars are moved in the queue as the time passes.

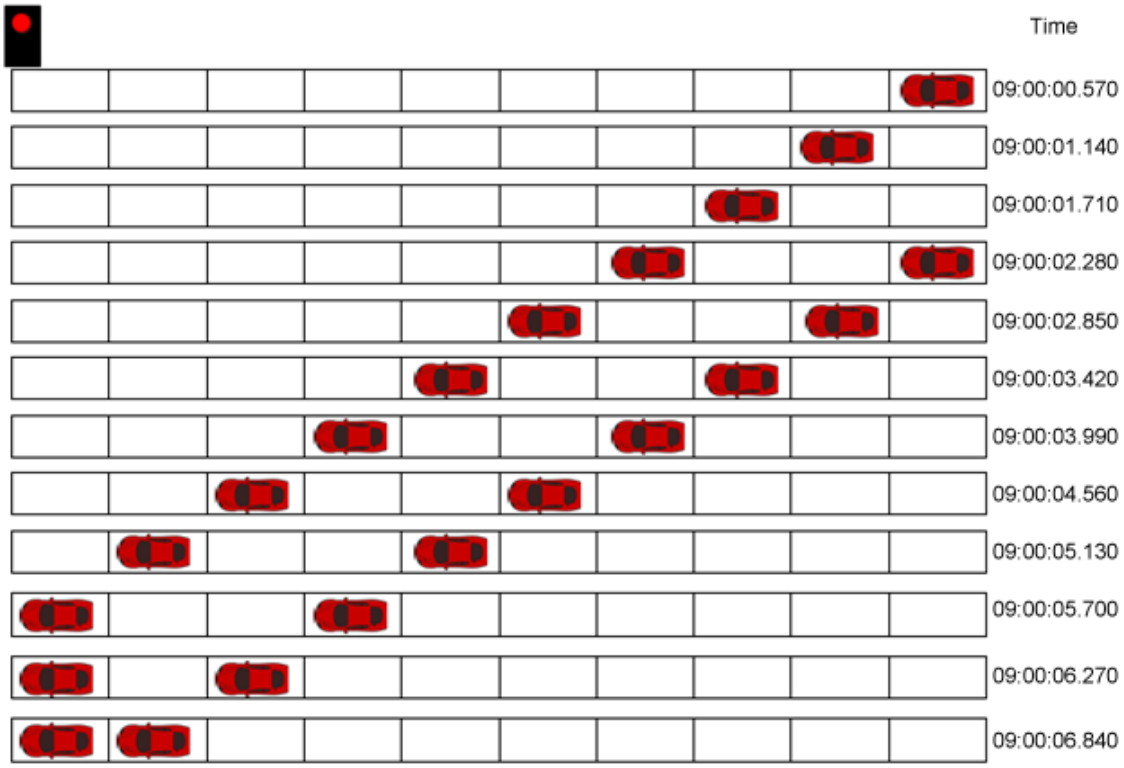

Figure 26: Movement of the Cars in CarQueue

7. Delayed Start

The agents in both the TCMS and the simulation system receive names of other agents in the systems which are needed for configuring the crossroads. Later they are used for the communication protocols. If an agent starts communication immediately after it is initialized, the communication may fail because the recipient is not initialized yet. That is why the communication between the agents should start after all the agents are initialized. In order to achieve this, the behaviour of the agents that initiates communication is delayed with three to five seconds. After five seconds all the agents in both systems should be created and initialized.

VI. Evaluation

The thesis provides the implementation of three different strategies for taking the decision when to change the lights of the semaphores. All of them implement the interface IScheduleCalculator and consist of one method which returns the calculated schedule. This chapter describes the strategies and compares their performance.

1. Strategy 1 – Fixed Intervals

The first strategy represents the usual situation in the real world – the semaphores have predefined plan, and they do not change it no matter what the traffic is. In the implementation the strategy is represented by the class FixedScheduleCalculator and always returns the same values for the red and green periods – 30 seconds. The value is optimal for the algorithm and is retrieved from several executions with different values.

2. Strategy 2 – Higher Level Longer Interval

The second strategy changes the schedule dynamically taking into account the traffic level of the semaphores in one crossroad. The semaphores of the crossroad are split in two groups – the semaphores in one group are in state *Green* at the same time. The algorithm takes the highest traffic in each group and gets the minimum time needed for it from the following values.

Traffic Level	Minimum Time (in seconds)
NoTraffic	0
Low	15
Normal	30
High	40
Intensive	60
Blocked	70

Figure 27: Minimum Time for Strategy 2

3. Strategy 3 – Interval Depending on All Groups of Semaphores

Similar to Strategy 2, the third strategy has minimum time for each traffic level of the semaphore groups, but it also takes into account the traffic level of both groups in the following way:

- Adds 10 seconds for each level above the *Normal* level of traffic of the current group of semaphores. For instance, if the traffic level is *Intensive*, it is two levels above level Normal and it adds 20 seconds to the minimum time required for *Intensive* level.
- Subtracts 5 seconds for each level above the *Normal* level of traffic of the opposite group of semaphores.

The minimum values for each traffic level are the following:

Traffic Level	Minimum Time (in seconds)
NoTraffic	0
Low	15
Normal	20
High	40
Intensive	60
Blocked	70

Figure 28: Minimum Time for Strategy 3

4. Comparison

In order to measure the performance of the system with the different strategies, the simulation has an output of log file containing the average time in seconds a car needs to pass one meter. This time is used to calculate the average speed of the cars in the city.

Since the generation of the cars is random, a simulation cannot be repeated twice. The Simulator generates cars with random directions but the number of cars in the same moment of two executions is the same. That is why the simulator provides more or less the same flow of cars for each execution.

In order to compare the strategies, the simulation is executed three times with each strategy with the same number of crossroads - 6. The following table contains the results of the execution.

Strategy	Seconds a car needs to pass 1 meter				Average Car Speed (km/h)
	First Execution	Second Execution	Third Execution	Average	
Strategy 1	0.148	0.151	0.152	0.149	24.16
Strategy 2	0.137	0.135	0.140	0.137	26.28
Strategy 3	0.133	0.132	0.133	0.133	27.07

Figure 29: Strategy Comparison

During the execution in all cases the number of the generated cars is the same – 3600. More or less in each attempt 3550 of these cars cross the city before the execution is terminated. The results of the execution show that the first strategy has the lowest performance. A car crosses the city with the average speed of 24.2 km/h, while in strategies 2 and 3 the speed is 26.3 km/h and 27.1 km/h. This means that with such a simple change in the algorithm for changing the lights, there is an improvement of the traffic control. Additionally, the initial values for each strategy are just a proposal. The can be adjusted to have optimal result.

Moreover, the strategies implemented in the thesis are very simple. More complex algorithms can be applied to increase the average speed of a car in the city. Another possible solution may take into account the time a car spends to pass from one semaphore to another and try to synchronize all the semaphores.

VII. Conclusions and Future Work

This thesis describes a distributed multi-agent system for traffic lights management. Following several methodologies for developing multi-agent systems, and using different notations, the analysis and design phases of the development process are explained in separate chapters.

The project is split in three parts – Traffic Control Management system (TCMS), Simulation system and graphical application. The Traffic Control Management system contains an independent system for managing the traffic lights.

The thesis proposes three different strategies for deciding the schedule of the lights change and compares them. As shown from the results the proposed strategies for the traffic management can improve the problem of the traffic by minimizing the time a car waits for a semaphore and increasing the average speed of the cars in the city.

The system is a simplified representation of the real world and has some limitations which can be improved in future work. In the current solution each street has only one lane in each direction and each semaphore always has four incoming and four outgoing lanes. Another possible improvement could be done in the simulation where the cars move with a constant speed, while in the real world each stopping of a car reduces the average speed.

From software engineering point of view, one of the main problems during the implementation of the system was the testing of the protocols. Unfortunately, there is no framework for testing agents and the communication between them. That is why any change of the protocols of the system will require a lot of work to assure that the system functions in the expected way.

VIII. References

[1] M. Wooldridge (2009). An Introduction to MultiAgent Systems

[2] Scott A. DeLoach & Mark Wood (2000). Multiagent Systems Engineering: the Analysis Phase.

[3] M. Wooldridge, N. R. Jennings, and D. Kinny (2000). The Gaia Methodology for Agent-Oriented Analysis and Design

[4]Lin Padgham and Michael Winikoff (2004). Developing Intelligent Agent Systems: A Practical Guide

[5]Federico Bergenti, Marie-Pierre Gleizes and Franco Zambonelli (2004). Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook

[6]AUML Notation, http://www.auml.org/

[7] Miguel-Angel Sicilia (2007). Competencies in Organizational E-learning: Concepts and Tools

[8] JADE v4.3.3 API, http://jade.tilab.com/doc/api/

[9] Giovanni Caire (2009), JADE Tutorial JADE Programming for Beginners

[10] Foundation for Intelligent Physical Agents Specification, http://www.fipa.org/specifications/

[11] R. Studer, R. Benjamins, and D. Fensel, (1998). Knowledge engineering: Principles and methods. Data & Knowledge Engineering